Adapting Distributed Scientific Applications to Run-Time Network Conditions*

Masha Sosonkina

Ames Laboratory and Iowa State University, Ames IA 50010
masha@scl.ameslab.gov

Abstract. High-performance applications place great demands on computation and communication resources of distributed computing platforms. If the availability of resources changes dynamically, the application performance may suffer, which is especially true for clusters. Thus, it is desirable to make an application aware of system run-time changes and to adapt it dynamically to the new conditions. We show how this may be done using a helper tool (middleware NICAN). In our experiments, NICAN implements a packet probing technique to detect contention on cluster nodes while a distributed iterative linear system solver from the pARMS package is executing. Adapting the solver to the discovered network conditions may result in faster iterative convergence.

1 Introduction

A typical high-performance scientific application places high demands on the computational power and communication subsystem of a distributed computing platform. It has been recorded [12] that cluster computing environments can successfully meet these demands and attain the performance comparable to supercomputers. However, because of the possibly heterogeneous nature of clusters, such performance enhancing techniques as load balancing or non-trivial processor mapping become of vital importance. In addition, the resources available to the application may vary dynamically at a run-time, creating imbalanced computation and communication. This imbalance introduces idle time on the "fastest" processors at the communication points after each computational phase. Taking into consideration an iterative pattern of computation/communication interchange, it could be beneficial for a distributed application to be aware of the dynamic system conditions present on the processors it is mapped too. Dynamic system conditions could include the load on the CPU, the amount of memory available, or the overhead associated with network communication. There have been many tools (e.g., [1, 5]) created to help learn about the conditions present. One of the requirements for such tools is to provide an easy-to-use interface to scientific applications, which also *translates* and *filters* the low-level detailed system information into categories meaningful to applications. For a scientific application, in which the goal is to model and solve a physical problem rather than to investigate the computer system performance,

* This work was supported in part by NSF under grants NSF/ACI-0000443, NSF/ACI-0305120, and NSF/INT-0003274, in part by the Minnesota Supercomputing Institute, and in part by the U.S. Department of Energy under Contract W-7405-ENG-82.

such an interface is very important. It also constitutes a major reason why inserting "performance recording hooks" directly into application's code may not be viable for a wide range of applications and for a multitude of computing system parameters. Thus the usage of a helper middleware is justified. When used at application's run-time, the middleware must be light-weight contrary to typical throughput benchmarks or operating system calls, which may heavily compete with the application for network or system resources.

In this paper (Section 3) we outline a network probing technique that, by means of sending carefully chosen trains of small packets, attempts to discover network contention without exhibiting the overhead inherent to network throughput benchmarks [9]. A light-weight technique may be used *simultaneously* with the computational stage of a scientific application to examine the communication subsystem without hindering the application performance. Section 2 presents a brief description of a proposed earlier framework that enables adaptive capabilities of scientific applications during the runtime. In Section 4, we consider a case study of determining dynamic network conditions while a parallel linear system solution solver pARMS is executing. A short summary is provided in Section 5.

2 Enabling Runtime Adaptivity of Applications

Network Information Conveyer and Application Notification (NICAN) is a framework which enables adaptation functionality of distributed applications [10]. The main idea is to decouple the process of analyzing network information from the execution of the parallel application, while providing the application with critical network knowledge in a timely manner. This enables non-intrusive interaction with the application and low overhead of the communication middleware. NICAN and the application interact according to a register and notify paradigm: the application issues a request to NICAN specifying the parameters it is interested in, and NICAN informs the application of the critical changes in these parameters. The application adaptation may be triggered by NICAN when certain resource conditions are present in the system. When the distributed application starts executing, each process starts a unique copy of NICAN as a child thread. This implementation is particularly useful in a heterogeneous environment because there is no requirement that the NICAN processes be homogeneous or even be running on the same type of machine. The process of monitoring a requested network parameter is separated from the other functions, such as notification, and is encapsulated into a module that can be chosen depending on the network type, network software configuration, and the type of network information requested by the application. Figure 1 depicts NICAN's functionality as the interaction of four major entities.

NICAN reads the resource monitoring requests using an XML interface, which enables diverse specifications and types of the application requests. Another functionality of NICAN is to call the adaptation functions provided when appropriate. If a particular resource condition is never met, then the adaptation is never triggered and the application will proceed as if NICAN had never been started. To make NICAN versatile and to provide a wide variety of monitoring capabilities, it is driven by dynamically loaded modules.

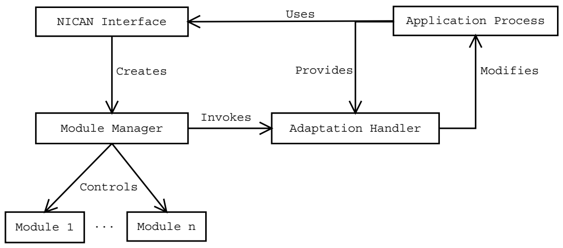

Fig. 1. Interaction of NICAN's components

```
/* Include the NICAN header file */
#include <nican.h>
/* The handlers are declared in the global scope */
void HandlerOne(const char* data) {/* empty */};
void HandlerTwo(const char* data) {/* empty */};
/* The application's main function */
void main() {
    const char xmlFile[] = "/path/to/xmlFile";
    /* Start NICAN's monitoring */
    Nican_Initialize(xmlFile,
                     2,
                     "HandlerOne", &HandlerOne,
                     "HandlerTwo", &HandlerTwo );
    /* Application code runs while NICAN operates */
    /* ... */
    /* Terminate NICAN's monitoring */
    Nican_Finalize(); }
```

Fig. 2. A trivial example of how to use NICAN

Figure 2 demonstrates how easy it is for the application to use NICAN. First the application must include a header file with the declarations for the NICAN interface functions. There are two adaptation handlers specified, *HandlerOne* and *HandlerTwo*, which for this trivial example are empty functions. The path to the XML file is specified and passed as the first parameter to *Nican_Initialize*. The application is informing NICAN of two adaptation handlers. *Nican_Initialize* will start the threads required for NICAN to operate and return immediately, allowing it to run simultaneously with the application. When the application is finished, or does not require the use of NICAN any longer, it calls the *Nican_Finalize* function, which returns after all the threads related to NICAN have been safely terminated.

3 Packet Probing to Examine Communication Overhead

In a cluster, we can consider two types of network transmissions following the terminology given in [2]. One type is *latency bound* transmission, and the other is a *bandwidth bound* transmission. A latency bound transmission is one where the transmission time required is dependent only on a one-time cost for processing any message. By using very small messages, on the order of a few hundred bytes, the cost of processing the message is reduced to a minimum. A bandwidth bound transmission is one where the time required is dependent on not just a one-time cost, but also the bandwidth available (see e.g., [9]). Typically bandwidth bound transmissions are comprised of large messages, which cause additional overhead while those messages are queued on network interfaces and processed. Latency bound transmissions have the attractive property that they do not cause a significant software overhead on the protocol stack of the machine processing the message. Thus, latency bound transmissions may be used as a means of communication subsystem performance analysis at application's runtime. This can be done while a distributed application performs its computational phase so as to not interfere with the application's communications. To better capture what conditions are present on a neighboring node, we use a train of packets rather than only two. This allows more time for the conditions to influence our train and is a trade-off between only two packets and flooding the network. We can use the notion of initial gap similar to the LogP model [3] to describe what happens to the probing packets. By introducing different types of load on the source and sink nodes we can affect the gaps recorded at the sink in a way that can be used to determine what type of load is present. The two metrics we will be using are the average gaps recorded at the sink and the standard deviation of those gaps.

3.1 Example of Probing: Fast Ethernet Networks

The cluster for our experiments is a collection of nine dual-processor nodes connected via 100Mbps Ethernet links by means of a shared hub. We send a train of packets directly from the source to the sink and perform timing on the sink. We vary the size of the packets across trials, but keep the number of packets constant. For each packet size we sent 100 trains of 64 packets and computed the average arrival time and the standard deviation for the distribution of arrival times. Note that 100 trains would be too many to be used in actual probing, as it is too close to "flooding" the network, but was used in our derivation of the technique described in this section. The actual amount of probing will depend on the time warranted by the computational phase of the application. We have conducted experiments that indicate using a single train of 32 packets may be sufficient. To add an additional load on the source and sink, we generated 30Mbps UDP network transmissions and external 98% CPU and memory loads on a node. Although we have performed a series of experiments to detect possible `(external load, competing flow)` combinations, we present here only a few illustrative cases. For more details see [11]. In particular, Figure 3 shows the case when no adverse conditions are present on the nodes. As packets grow larger they introduce a larger load on the network stacks. This is clearly the case with Figure 3 depicting a distinct *bend* formed at the point where the probes are 400B in size. We can approximate this point using the rate of increase for the arrival gaps as the probes get larger. A different situation is shown in Figure 4 where there is no pronounced bend. The "strength" of this bend can be used to help classify what conditions may be present. The location of the bend can be determined during the first set of probes, or by executing the proposed algorithm on a given computing system *a priori* before the actual application is run. Once this point is known we can narrow the scope of the probe sizes to reduce the impact and time required for the dynamic analysis.

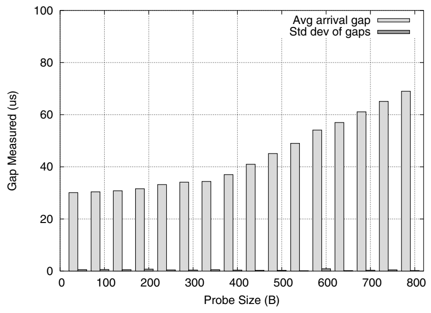

Fig. 3. No competing CPU load or network flows on either node

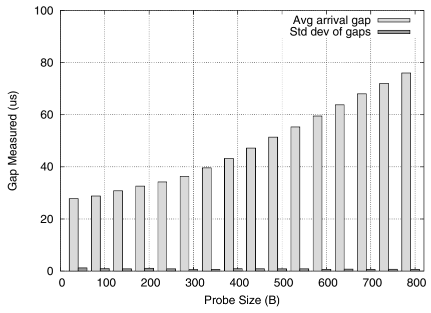

Fig. 4. No CPU load and 30 Mbps flow leaving the source node

In each experiment the gap introduced on the source between the packets, at the user level, was constant with very little deviation. Therefore any deviation measured at the sink was due to an external influence on our probes. Depending on what type of influence is placed on the source and sink we observe different amounts of deviation, which we use as the second factor to classify the network conditions present. In particular, for a given network type, we have constructed a decision tree (see [11]), tracing which one may detect seven possible cases of network or CPU related load on the source and sink nodes based on recorded deviations and average gap measurements.

4 Case Study: Runtime Changes in Communication Overhead of pARMS

pARMS is a parallel version of the Algebraic Recursive Multilevel Solver (ARMS) [8] to solve a general large-scale sparse linear system $Ax = b$, where A is a constant coefficient matrix, x is a vector of unknowns, and b is the solution vector. To solve such a system iteratively, one *preconditions* the system of equations into a form that is easier to solve. A commonly used (parallel) preconditioning technique, due to its simplicity, is Additive Schwarz procedure (see, e.g, [7]). In the iteration i, $i = 1, \ldots, m$, given the current solution x_i, Additive Schwarz computes the residual error $r_i = b - Ax_i$. Once r_i is known, δ_i is found by solving $A\delta_i = r_i$. To obtain the next iterate x_{i+1}, we simply compute $x_{i+1} = x_i + \delta_i$ and repeat the process until $|x_{i+1} - x_i| < \epsilon$, where ϵ is a user defined quantity. The Additive Schwarz procedure was used for all the experiments discussed here. To solve linear systems on a cluster of computers it is common to partition the problem using a graph partitioner and assign a subdomain to each processor. Each processor then assembles only the local equations associated with the elements assigned to it.

Our motivation for using packet probing is to find congested links in the underlying network of a cluster and to alert pARMS whenever its own adaptive mechanisms need to be invoked. To achieve this goal we have developed a module for NICAN that performs packet probing using the techniques described in Section 3. The design of MPI [4] allows pARMS to start a unique instance of NICAN in each task, each of which sends probes independently. Discovering network overhead on neighboring nodes has proven useful [6] for the overall performance of pARMS. Thus, ideally, we wish to have each node learn the conditions of the entire system.

We will demonstrate this NICAN module using a four-node pARMS computation, with each node probing a fifth node free of any network overhead. By using the various options provided by the module this situation is easily created using an XML file. The 4pack cluster with 32 dual Macintosh G4 nodes, located at the Scalable Computing Laboratory in Iowa State University, has been used for the experiments. Sixteen 4pack nodes have a single 400MHz CPU and the remaining have dual 700MHz CPUs. We used the faster nodes with Fast Ethernet interconnection for both our probes and MPI traffic because an implementation of NICAN packet probing module on Ethernet networks is already available. Figure 5 illustrates how the experiment is laid out. The squares represent the nodes used, with the label indicating the hostname of the machine. pARMS is mapped to `node0`,. . .,`node3`; the competing flows (called `Iperf traffic`) are entering node `iperf_dest`; and the probes sent from the pARMS nodes, are sinking into `probe_sink`.

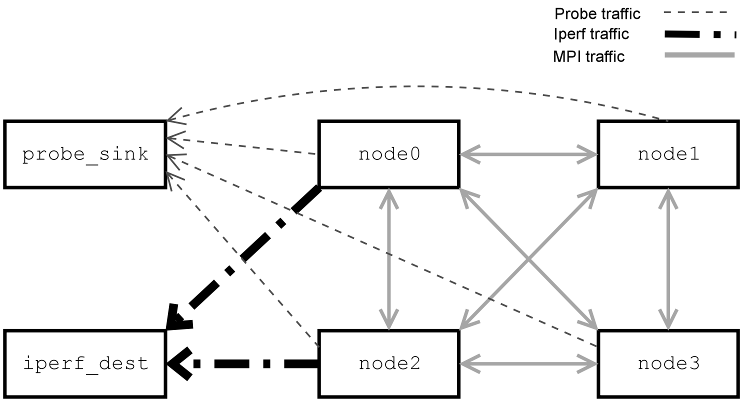

Fig. 5. Cluster node interaction used for the experiments

Consider the elliptic partial differential equation (PDE)

$$-\Delta u + 100\frac{\partial}{\partial x}\left(e^{xy}u\right) + 100\frac{\partial}{\partial y}\left(e^{-xy}u\right) - 1,000u = f \qquad (4.1)$$

solved on a two-dimensional rectangular (regular) grid with Dirichlet boundary conditions. It is discretized with a five-point centered finite-difference scheme on a $n_x \times n_y$ grid, excluding boundary points. The mesh is mapped to a virtual $p_x \times p_y$ grid of processors, such that a subrectangle of $r_x = n_x/p_x$ points in the x direction and $r_y = n_y/p_y$ points in the y direction are mapped to a processor. In the following experiments, the mesh size in each processor is kept constant at $r_x = r_y = 40$. In our experiments, four processors ($p_x = p_y = 2$) have been used, thus resulting in a problem of the total size 6,400. This problem is solved by FGMRES(20) using Additive Schwarz pARMS preconditioning with one level of overlap and four inner iterations needed to solve the local subproblem with GMRES preconditioned with ILUT (see [7]).

Each MPI node will use NICAN's packet probing module to send probing packets to a fifth node known to be free of any network load. Then, at pARMS runtime, any network flow detected is indicative of a load on a node involved in the computation. The experimental procedure is outlined in Table 1. The "Phase" column represents the different phases of the experiment an the "Conditions present" column details what conditions were present in addition to the pARMS-related traffic. Table 1 also lists the average times required to complete the distributed sparse matrix-vector multiplication (called SpMxV) during each phase of this experiment (see columns `node0`,. . .,`node3`). The impact of competing network traffic is evident in how the unaffected nodes spend more time completing SpMxV. The extra time is actually accrued while they wait for the node affected by the network flow to transmit the required interface variables. The affected node does not perceive as long a waiting time because when it finally requests the interface unknowns from a neighbor, that neighbor can immediately send them. When we introduce the two network flows at the phase p_2, both `node0` and `node2` experience less waiting time, but we can also see how much impact competing network traffic can exert on pARMS. The average waiting time for unaffected nodes in this case has nearly tripled compared with p_0, increasing the overall solution time as a result. We only show the times for SpMxV because that is the specific source of increased execution time when additional network traffic is exerted upon a node in the computation. Once we are able to detect and monitor how much waiting time is incurred by each node a suitable adaptation can be developed to help balance the computation, and improve the performance in spite of competing network traffic [6]. The gist of the pARMS adaptation procedure is to *increase* the number of inner Additive Schwarz iterations performed locally in the *fast* processors, i.e., in those processors that incur idle time the most. For example, `node1` and `node3` are such processors in phase p_2. The amount of increase is determined experimentally and may be adjusted on subsequent outer iterations, such that the pARMS execution is kept balanced. With more inner iterations, the accuracy of the local solution becomes higher and will eventually propagate to the computation of the overall solution in an outer (global) iteration, thus reducing the *total number* of outer iterations. Figure 6, taken from [6] for the measurements on an IBM SP, shows the validity of suggested pARMS adaptation. In particular, with the increase of the number of inner iterations, the waiting time on all the processors becomes more balanced, while the total solution time and the number of outer iterations decrease. Figures 7 and 8 demonstrate how the gaps recorded on the nodes are used to determine the conditions present. In Figure 7 the bend discussed in Section 3 is clearly visible when the probes transition from 350B to 400B. Also, because we are using only 32 probing packets the deviation is larger than that observed in Figure 3. In Figure 8 we see the effect that the competing flow has on the probing packets. The distinct bend is no longer present, and by tracing through the decision tree corresponding to the given network type, we can determine that there is a competing flow leaving the source. We only illustrate the results for these two cases but similar plots can be constructed to show how the other conditions are detected.

Table 1. Phases of network conditions and average times (s) for SpMxV at each phase during pARMS execution

Phase	Conditions present	node0	node1	node2	node3
p_0	No adverse conditions present	.0313	.0330	.0331	.0398
p_1	40 Mbps flow from node0	.0293	.0698	.0899	.0778
p_2	40 Mbps flows from node0 and node2	.0823	.1089	.0780	.1157
p_3	40 Mbps flow from node2	.0668	.0847	.0291	.0794
p_4	No adverse conditions present	.0293	.0319	.0474	.0379

Adapt. yes/no	Outer Iter.	Solution, s
no	5,000	4,887.78
yes	432	398.67

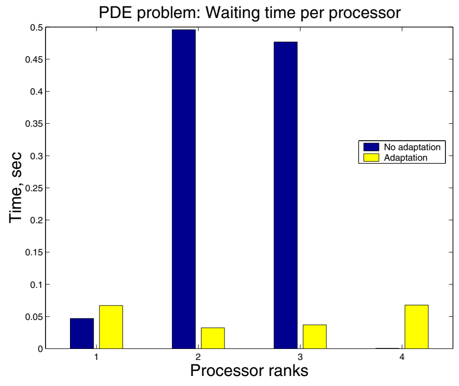

Fig. 6. Adaptation of pARMS on a regular-grid problem

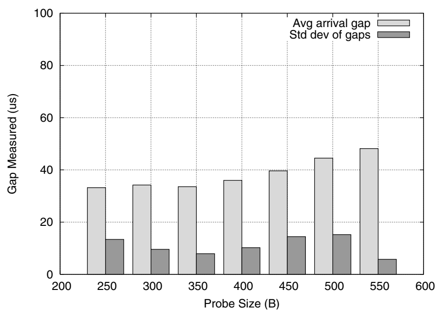

Fig. 7. The probe gaps observed by node0 at phase p_0 in Table 1

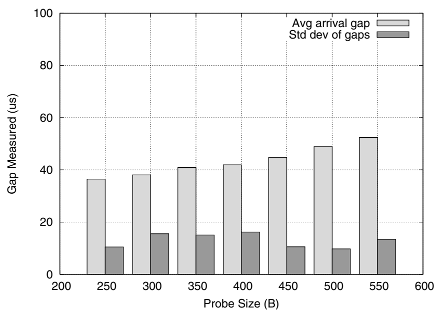

Fig. 8. The probe gaps observed by node0 at phase p_1 in Table 1

5 Conclusions

We have described a way to make distributed scientific applications network- and system-aware by interacting with an easy-to-use external tool rather than by obtaining and processing the low-level system information directly in the scientific code. This approach is rather general, suiting a variety of applications and computing platforms, and causes no excessive overhead. The case study has been presented in which the NICAN middleware serves as an interface between parallel Algebraic Recursive Multilevel Solver (pARMS) and the underlying network. We show how a light-weight packet probing technique is used by NICAN to detect dynamically network contention. In particular, NICAN is able to detect and classify the presence of competing flows in the nodes to which pARMS is mapped. Upon this discovery, pARMS is prompted to engage its own adaptive mechanisms leading to a better parallel performance.

References

1. D. Andersen, D. Bansal, D. Curtis, S. Seshan, and H. Balakrishnan. System support for bandwidth management and content adaptation in Internet applications. In *Proceedings of 4th Symposium on Operating Systems Design and Implementation*, pages 213–226, 2000.
2. C. Bell, D. Bonachea, Y. Cote, J. Duell, P. Hargrove, P. Husbands, C. Iancu, M. Welcome, and K. Yelick. An evaluation of current high-performance networks. In *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS'03)*, 2003.
3. D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Principles Practice of Parallel Programming*, pages 1–12, 1993.
4. Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report Computer Science Department Technical Report CS-94-230, University of Tennessee, Knoxville, TN, May 5 1994.
5. J. Hollingsworth and P. Keleher. Prediction and adaptation in Active Harmony. *Cluster Computing*, 2(3):195–205, 1999.
6. D. Kulkarni and M. Sosonkina. A framework for integrating network information into distributed iterative solution of sparse linear systems. In José M. L. M. Palma, et al. editors, *High Performance Computing for Computational Science - VECPAR 2002, 5th International Conference, Porto, Portugal, June 26-28, 2002, Selected Papers and Invited Talks*, volume 2565 of *Lecture Notes in Computer Science*, pages 436–450. Springer, 2003.
7. Y. Saad. *Iterative Methods for Sparse Linear Systems, 2nd edition*. SIAM, Philadelpha, PA, 2003.
8. Y. Saad and B. Suchomel. ARMS: An algebraic recursive multilevel solver for general sparse linear systems. Technical Report Minnesota Supercomputing Institute Technical Report umsi-99-107, University of Minnesota, 1999.
9. Q. Snell, A. Mikler, and J. Gustafson. NetPIPE: A network protocol independent performance evaluator. In *IASTED International Conference on Intelligent Information Management and Systems*, June 1996.
10. M. Sosonkina and G. Chen. Design of a tool for providing network information to distributed applications. In *Parallel Computing Technologies PACT2001*, volume 2127 of *Lecture Notes in Computer Science*, pages 350–358. Springer-Verlag, 2001.
11. S. Storie and M. Sosonkina. Packet probing as network load detection for scientific applications at run-time. In *IPDPS 2004 proceedings*, 2004. 10 pages.
12. Top 500 supercomputer sites. http://www.top500.org/.